多核/众核系统上的并行编程语言

陈莉 霍玮 卢兴敬 唐生林

1 引言

半导体技术的不断进步使得单位芯片面积集成的晶体管数目不断增长。但是由于功耗、复杂性等方面的限制，传统的通过提高主频、增加结构的复杂度来追求性能提升的道路，已经不能充分利用丰富的晶体管资源提高应用程序的性能。业界转变了体系结构的设计思路，提出多核/众核技术，通过提供丰富的片内并行性（包括线程级并行和数据并行等）来提高芯片的处理能力。目前，多核处理器已逐渐成为市场上主流的通用芯片，进行面向多核/众核的编程方式的研究势在必行。

编程模型是保障易编程性和高效性的前提，是体系结构和应用之间的桥梁，向上隐藏运算、存储及互连等体系结构的细节，同时提供编程所需的硬件抽象支持；向下充分利用硬件提供的资源，高效正确地完成软件操作。目前，针对片上并行系统的编程模型和语言的研究正处在百花齐放的阶段。编程模型和语言有的是为某种体系结构量身定制（如 CUDA，HMPP），或是适应某种应用领域的迫切需求（如 ErLang, StreamIt）；编程模型可以以编程语言的形式提供给程序员来表达并行，也可以使用库、模板等形式，多数是在现有编程语言上添加语法制导或扩展少量关键字；还有的提供一个并行框架，程序员使用现有语言编写核心的功能模块，嵌入并行框架并行执行。

与过去相比，多核、众核平台上并行编程面临着若干新的挑战：

– 应用的行为特征更加复杂、应用有新需求

传统的并行编程语言面对的是少数专家用户，编程层次低、产能低，常常只适合表达静态的粗粒度并行性。而在多核普及的今天，应用往往要面对动态不规则的数据结构，程序的并行性也具有更大的不规则性和动态性，比如出现在 `while` 循环中、结构化的代码段之间、递归等控制结构上。编程语言需要发掘各种非静态的并行模式，比如流水线并行、任务树、任务图等。此外，数据中心已成为和水、电一样重要的基础设施，越来越多的应用需要满足高度并发负载和海量数据处理的需求，这给编程语言提出新的需要。

– 可编程性

大多数已有软件都是串行程序，并且规模非常庞大，分布在桌面、嵌入式以及服务器领域。如何以最小的代价修改当前已有的串行代码使其能够在并行系统上实现最大的性能提升成为业界最为关心的问题。研究具有后向兼容性的语言扩展，提出有效的工具和技术以支持可实际应用的安全的并行化是人们所期待的。

对于新的并行应用，用户希望以较高的抽象层次开发并行程序，尽可能少地参与并行调度、存储管理、数据同步和通信，并减少并发和并行相关的错误，比如数据竞争、原子性违反、死锁、顺序违反等。

语言设计方面主要有两类对策：(1). 提供隐式并行性的表达。比如，提供基于高级数据

类型的并行操作（如向量操作）、面向领域的语言和框架；(2). 提供各种语言层或者库的高层抽象，简化并行性/局部性的描述和映射，同时避免某些并发错误的发生。具体包括，存储和计算的逻辑单位和物理单位分离，提供丰富的并行模板或并行结构，提供高层的同步机制，提供线程安全的并发容器和函数式语言特征等。

– 复杂的存储和互连结构

多核体系结构下，各个部件之间通过互连网络来实现数据通信。但通信的带宽、延迟以及同步方式存在差异。为了满足众多小核对于访存延迟和带宽的需求，存储系统通常具有更加复杂的层次特性。如何利用好这些容量不同、速度各异以及具有不同共享特性的存储空间，并通过通信延迟的消除或隐藏、流量控制、带宽以及通信同步方式等的高效优化利用，对于程序的性能有非常大的影响。

特色加速部件是对某类算法进行优化，是提高应用程序性能、降低能耗的一种方法。这些特色加速部件包括支持通用计算的图形处理器（General Purpose Graphic Processing Unit，GPGPU）、IBM 的 CELL 芯片、数字信号处理器（DSP）、网络处理器(Network Processor)等。通用 GPU 的一些实例包括 Nvidia 和 AMD 生产的 GPU（集成或者独立的）、嵌入式领域的 PowerVR 移动图形核心、英特尔的 Knights Ferry 等。通用 GPU 常作为协处理器，具有与主 CPU 不一样的指令集和应用程序二进制接口（Application Binary Interface，ABI）、分离的存储，构成了计算平台上分布式的异构存储空间。因此程序员需要确定数据在各个存储空间的分布，任务在不同计算单元上的分配和调度，管理存储空间的分配与释放，确定数据传输的时机和优化以及维护数据的一致性，这些工作给程序员带来非常大的挑战。

多核系统上数据移动是影响性能和功耗的重要因素。一些编程接口和语言(如 MPI 和 PGAS)采用显式的数据分布，而另一些语言（如 TBB）通过任务调度和隐式数据分布的结合来实现数据重用，还有一些语言（如 sequoia）则不仅考虑水平方向的数据布局，也管理垂直方向的数据移动。

– 并行体系结构的多样性与应用的性能可移植性

通用编译器产生的代码通常不能有效地利用处理器资源，而程序员手工调优又需要花费大量的时间和精力。在这样的背景下，出现了各种辅助程序员对应用程序和库进行自动调优的技术。多核体系结构的设计呈现出复杂和多样性，使程序员在编写程序、优化性能、进行跨平台移植时面临更多压力和困难。如何在并行编程语言的设计中支持自动性能调优是实现跨平台的高性能可移植性的重要课题。

本文将从遗产代码的并行化、单程序流多数据流（SPMD）的并行编程接口、任务并行的语言特征、异构平台的编程框架、高并发的函数式语言和分割全局地址空间（PGAS）语言等方面，介绍多核和众核体系结构上并行编程语言和接口的现状和趋势。

2 遗产代码的并行化

本小节介绍两种并行化的方法，一个是以制导为主的 OpenMP 语言，一个是以编译技术为主导的 DSWP 模型。

2.1 OpenMP

1997 年颁布的 OpenMP 标准是 SMP[1]体系结构发展的结果。语言的初衷是针对稠密的数值计算应用的并行化问题，所以主要围绕着以数组访问为主的并行循环。到今天，OpenMP 已经成为共享内存体系结构上被广泛应用的并行编程语言。2005 年 5 月，OpenMP2.5 统一了 Fortran 和 C/C++的标准规范。2008 年，OpenMP 推出了 3.0 的语言规范，引入了任务的概念，支持非规则的循环并行、树状任务图的并行。OpenMP 作为一个事实上的工业标准，得到了如 GCC、PGI、微软 VS2005、Open64、Sun 等主流编译器和 Totalview、PGDBG 等调试器的广泛支持。

OpenMP 的机器抽象是对称多处理机，采用 fork-join（并行块） 的执行模型。并行域是 OpenMP 语言的基本并行结构，由当前线程创建一个线程组，且自己成为主线程启动并行执行。并行域的结尾处有隐式的栅障（barrier）同步。OpenMP 提供了以工作共享为基础的循环并行、静态任务划分的支持，并在 OpenMP3.0 以后提供动态的显式任务[5]，支持非计数循环和递归等不规则并行性。OpenMP 提供放松的内存一致性，用户负责描述并发单位（比如 workshare（工作分担）结构、任务（task）结构）的数据环境（私有或者共享）。

任务概念扩大了 OpenMP 的适用面。任务是当线程遇到 task 结构或者并行结构时生成的，包括了可执行代码和对应数据环境。并行结构中会生成一组隐式任务，每个任务被调度到一个线程。显式任务是可推迟执行的工作单元，它在同步点（barrier 和 taskwait）和任务区的开头结尾处进行动态任务调度。task 被调度到当前绑定的线程组上，映射一个新任务的执行或者恢复一个挂起状态的任务执行。图 1 示例了如何利用显式任务制导进行树遍历。需要注意的是，如果没有 taskwait 制导，这些并行任务的执行顺序将没有任何约

```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node
*);
extern void postorder
_traverse(struct node *);
void traverse( struct node *root){
  #pragma omp parallel
    #pragma omp single
        postorder_traverse
        (root);
}
```

```
void postorder_traverse( struct
node *p ) {
  if (p->left)
    #pragma omp task
    // p is firstprivate by
    default
    postorder_traverse(p->left)
    ;
  if (p->right)
    #pragma omp task // p is
    firstprivate
    postorder_traverse(p->right
    );
  #pragma omp taskwait
  process(p);
}
```

图1. 利用 OpenMP 的显式 task 进行树遍历

束。Single 制导是为了保证每个任务只派生一次。并行结构的数据共享属性有三个决定的层次：预先确定、显式确定、隐式确定。任务概念引入后，这个规则也变得愈发复杂。根据 OpenMP3.0 的规定，这里 p 的共享属性是 firstprivate。task 可以嵌套形成树型层次，产生动态不规则的细粒度任务，由于 task 的调度开销比较大，西班牙的巴塞罗那超级计算中心提出了一个 final 子句[3]来控制孩子 task 的融合。

1 Symmetric Multi-Processing，对称式多重处理架构

随着共享处理器上并行性的增加，OpenMP 原有的一层循环并行制导已经不能满足需求（比如通用 GPU 上提供了大量的并行性），OpenMP3.0 在 `for` 结构上增加了 `collapse` 子句使其能对多层嵌套循环进行并行化。

OpenMP 的机器抽象是对称的多处理器，不能适应通用 GPU 和嵌入式 MPSOC 平台的分离存储的特性。OpenMP 语言规范组织正在考虑引入 PGI Accelerator[2]针对通用 GPU 的语言制导经验。 文献[1]则认为动态数据流模型更适合分布存储的系统或者异构的系统。它借鉴了 StarSs 的动态机制，在 `task` 上增加数据输入、输出的子句来描述指定 `task` 之间的依赖关系，增加以数据为中心的 `task` 同步制导，并增加设备映射的子句。

OpenMP 在共享内存的服务器上具有较高的普及率，但它也有一些明显的不足。由于不能表达数据的局部性，也由于执行模型过多地依赖全局同步，OpenMP 的性能可控性和扩展性都不够好。虽然原则上，OpenMP 只需对串行程序作少量改动就可以实现程序的并行化，但是对于一些复杂的应用，为了设置正确的数据共享属性，可能需要大量修改原有的数据结构，并同时带来数据竞争等难以排查的错误。为了帮助用户编程，各个硬件厂商都会给 OpenMP 编译器提供一些正确性检查工具，比如英特尔的 Parallel Inspector（并行检查器）（以前叫 thread checker（线程核对器））。在可编程性方面，OpenMP 在应用适用面方面受到众多新兴语言的挑战。如何能更好地适应新兴的多核、众核平台，也会在很大程度上影响其应用前景。

2.2 DSWP 模型

DSWP[2]是普林斯顿大学的计算机科学和电子工程系的学者于 2005 年左右推出的一种遗产代码的并行化技术。流水是一种在处理器结构和编译器指令集并行优化中广泛使用的模型，普林斯顿大学将其推广应用到线程级并行上，提出 DSWP 这种并行执行模型。并主张通过编译分析技术自动划分程序中的流水段，并产生流水段之间的通信，达到对非规则程序自动并行化的目的。

```
cost = 0;
node = list->head;
while(node) {
   ncost = doit(node);
   cost += ncost;
   node = node->next;
}
```

(a)循环

(b) 程序依赖图

箭头表示依赖关系，虚线表示形成依赖环

图2. DSWP 并行循环

DSWP 目前主要是针对 C 程序中的循环，允许有非规则的数据结构和较复杂的控制流。图 2（a）中所示的一个例子是如何针对 `while` 循环，应用 DSWP 技术对循环进行并行化。编译器首先分析各语句间的依赖关系，构造出程序依赖图（Program Dependence Graph, PDG），程序依赖图中包含了所有语句间的数据依赖和控制依赖，如图 2（b）所示。然后编译器在程序依赖图中构造强连通分量，将形成依赖环的语句聚合到一起，强连通分量便是分

[2] Decoupled software pipelining，解耦软件流水线

配线程的基本单元。将程序依赖图中的每个强连通分量合并成一个单节点，这样的程序依赖图称为 DAG_{SCC}。所有的强连通分量按照一定的策略划分到多个线程上，所有的线程形成一条流水线，划分时仍然要满足流水段之间不能形成依赖环。图 3 示意了一种可能的映射：

图 3 中有 3 个强连通分量，映射到两个线程上。每个线程相当于一个流水段。在 DSWP 模型中，映射到同一线程中的语句自然满足了串行语义的依赖关系（如语句 3 和 6），不必产生通信代码。映射到不同线程的语句间由于并行执行因此要产生通信（如语句 3 和 4，3 和 5），通信的代码由编译器负责插入。DSWP 模型的一个重要好处是线程间通信沿流水线方向单向流动，不会形成回路，这样就避免了由于等待通信而造成的流水线停滞。

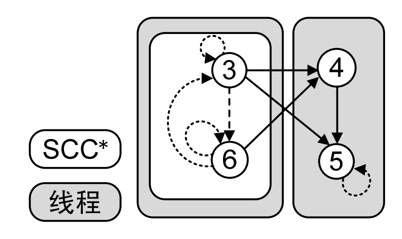


图3. DSWP 构造强连通分量和线程映射示意

但是 DSWP 依赖通信队列的高效实现，甚至需要专门的硬件支持(`produce` 和 `consume` 指令)。

在以上介绍的 DSWP 基本模型中，并行度受制于执行最慢的流水段。另外，如果强连通分量的数目少于可用核（线程）的数目，也不能充分利用并行资源。因此普林斯顿大学的研究人员进行了改进，对流水段进行复制以提高并行度，称为 Parallel Stage DSWP (PS-DSWP)[7]。

研究人员为了辅助编译器挖掘出更多的并行性，提出了在 C 语言中增加两个制导："`commutative`"和"`ybranch`"**[6]**。"`commutative`"标注一个函数，表明该函数的执行次序是可交换的，即并行时可以忽略此函数带来的依赖。"`ybranch(probability)`"标注一个条件分支，提示该条件在一定的频率下向"**`TRUE`**"方向跳转。这一方法目前看到的应用范围还不十分广泛。

流水线模型是一种很有效且广泛使用的执行模型。DSWP 的研究人员主张通过编译器自动分析程序，按照语句间的依赖关系把程序划分成若干流水段，然后把流水段映射到线程上。这为并行化提供了一种很好的思路。编译器可以挖掘多重粒度的并行段并按一定策略进行组合，从而有机会结合芯片结构做更多性能优化的考虑。DSWP 的主要局限是对编译器的要求极高，现有的编译器对 C/C++分析的精度还并不理想，数组、指针运算等都是对编译分析的障碍。目前发表的结果基本都是基于手工并行变换。自动变换是否能成为实用系统还难以判断。

3 消息传递接口——MPI

消息传递接口 MPI[3]创建于 1992 年，它是针对分布主存的可扩展并行计算机（比如集群）而设计的编程接口，目前已经成为这类系统上高性能并行程序设计的事实标准。MPI 以库的方式支持进程间的数据交换，提供 Fortran 和 C 的编程接口。MPI 的典型实现包括开源的 MPICH、LAM MPI 以及不开源的 Intel MPI 和 HP MPI 等。2009 年 MPI-2.2 标准发布。

MPI 表达的是单程序流多数据流的粗粒度并行性，它以通信域的方式支持进程分组，并表达进程间的拓扑组织。在通信域之上，建立了丰富的显式的点到点通信机制，包括阻塞、

3 Message Passing Interface，一种基于消息传递的并行程序设计标准

非阻塞通信，双边或者单边通信，多种通信模式等；针对科学计算应用，它特别提供了丰富的集合通信操作。MPI 程序具有很好的数据局部性，某些实验表明，在多核结构[14]上 MPI 程序在大部分情况下性能好于 OpenMP 和 UPC。

3.1 MPI 对通用多核系统的支持和优化

随着多核的普及和发展，片上核数越来越多，MPI 和线程编程的混合是一个有效利用多核同时提高应用性能扩展性的方法。目前，MPI Forum 正在讨论的 MPI-3[8]特别关注了 SMP 与多线程的混合编程，支持 MPI+X 的编程方案，提供了 MPI 与 X 之间的外部调用接口，X 可以是 Pthreads、OpenMP 或者新兴的编程接口如 TBB、 OpenCL、 CUDA、Ct 以及 UPC 等分割全局地址空间（PGAS）语言。混合编程方案能充分发挥 MPI 和 X 两种编程模型各自的特长，整体上可获得良好的性能。但是，其易编程性比较差。这里同时要解决的一个问题是如何使 MPI 实现高效地支持多线程之间的消息并发。MPI 的线程安全仍存在性能问题——多数 MPI 实现不是线程安全的。文献[15]研究了降低临界区粒度，提高线程安全的 MPI 的实现性能的方法。文献[9]、[10]提出 TMPI，其中每个 MPI rank 是一个线程，采用编译变换消除全局量，并结合运行时通信优化技术，实现了将同一 SMP 节点内的多个 MPI 节点映射到同一进程内的多个线程，获得了较高的 MPI 性能，特别是在多道程序运行环境下性能突出。

有不少工作涉及 MPI 在 SMP 节点内的性能加速。比如，文献[11]通过修改操作系统页表映射策略使 MPI 进程除了拥有私有地址空间，还可以自由读写同一 SMP 结点的其它进程的地址空间，借助这样的策略提高了 MPI 集合通信的性能。但这方法依赖于具体的硬件和操作系统，无法适用于更多的场合。

3.2 嵌入式多核的通信标准 MCAPI

目前，嵌入式处系统普遍采用多核架构，处理器或者核之间常常没有存储一致性，存在着多个维度上的异构，包括核异构、互联异构、存储异构、OS 异构、软件工具链异构和编程语言异构等。这对嵌入式多核系统的应用编程带来了极大的挑战。为了提高开发这类系统应用的产能，必须对体系结构进行抽象，向应用层提供统一的编程接口。为此多核联盟[13]制订了嵌入式系统中相近分布的核和/或处理器间的通信和同步的标准化应用编程接口 MCAPI。该接口独立于语言、处理器和操作系统，其最新版本为 V2.015，基本目标是提供极高的性能，减小访存足迹。Poly-Messenger®/MCAPI 是多核联盟 MCAPI 的一个商业化实现。

MCAPI 利用多核芯片互联网络所提供的低时延通信接口，提供三种通信模式：消息、包和标量，并在负载的灵活性，允许动态改变接收者、设置优先级以及配置等方面提供了灵活的选择。MCAPI 支持各种各样的服务质量，每条消息都能有一个优先级，允许将部分或全部通道映射到硬件上，也能通过对有连接的通道指定属性来支持零拷贝。与 MPI 相比，MCAPI 在扩展性和容错方面有相似的目标，但有更强的约束，在应用领域方面通用性更强，灵活性上稍差。遵照 MCAPI 规范实施嵌入式系统应用的开发大大提高了可移植性。

3.3 成功经验

深入思考 MPI 的成功现象，对于设计新的实用的并行程序设计语言有很大的借鉴意义。MPI 的成功来自以下 8 条优势：(1). 语言规范开放；(2). 可移植性非常好；(3). 在存储层次的管理、组通信等方面具有高性能；(4). 简单：MPI-2 只有 275 个函数，对于一般的 MPI 程序，只需要很少的库函数就够了。MPI 在概念引入上也非常精简，比如通信域和数据类型

的概念处理；(5). 结构模块化：通信域的封装机制支持基于组件的软件开发、支持专用库的构造，从而使消息传递的应用程序中可以不直接调用 MPI 函数；(6). 互操作性好。可以与编译器、调试器、profiler[4]、多线程程序有效地实现互操作；(7). 语义完备。表达能力非常强，能描述任意复杂的并行程序行为；(8). 性能透明。

可以预见的是，MPI 将继续流行。在 Fortran 和 C 语言之外，已经出现了很多非传统的 MPI 实现语言，支持 Python, Ruby, Java 以及微软的 CL。但是，MPI 程序的开发效率较低，且单程序流多数据流的执行模型与多核体系结构不匹配，并不是理想的编程语言。

4 任务并行的支持

并行编程语言的一个重要发展趋势是用逻辑任务代替线程，实现对应用程序中计算的虚拟化。异步任务的提出改善了编程效率，也成为实现容错的重要手段。同时，并行编程语言有自己的运行时系统，负责用户空间上的任务调度，而操作系统只负责内核空间上的线程调度。这样，并行语言的运行时能利用应用程序的高层信息进行有效的调度。此外，任务窃取等任务调度技术也有助于应用程序的负载平衡。我们介绍两个典型的语言：TBB 和 Sequoia。前者是一个并行库；后者则基于一个统一的机器抽象，以实现跨平台的性能可移植性。

4.1 TBB 与 TPL

Intel® Threading Building Blocks (TBB，线程构建模块) [16]是由英特尔公司开发的一个并行库，它运行于共享存储的多核平台，可以在 Windows、Linux 和 Mac OS X Systems 上运行。TBB 是一个 C++的并行库，通过提供并行算法模板、并行容器、同步原语、可伸缩的内存分配函数来支持高效的并行。TBB 中使用逻辑并行结构代替线程，并利用并发集合和并行算法来支持并行，提高了可编程性。

另外，微软.NET 4.0 新提供了一个并行线程库 Task Parallel Library(TPL，任务并行库)，TPL 与 TBB 类似，表达形式和任务调度算法都与 TBB 基本一致，此处就不再单独介绍了。

TBB 给用户提供一套并行算法或并行模板，调度算法负责负载平衡和高效利用缓存（cache），使编写并行程序变得容易。图 4 是使用 TBB 编写并行求和程序的代码示例。`parallel_for` 是一个并行模板，在一个可划分的迭代空间(range)上进行并行的归约操作。归约操作的具体行为由对象 `sf` 指定，对应的类必须包括：局部归约、全局归约（`join`）、析构、分裂构造四个方法。这里 `blocked_range` 模板是一个可以进行递归划分的半开区域。实际上，高层的循环模板中隐含着任务调度器，隐藏了调度器的复杂性。TBB 除了提供传统的循环并行的模板，也支持流水线等常见的并行模式。

如果程序员无法用高层模板表达某个算法，则可以利用任务组的支持，或者调用任务调度器的接口函数，来使这个算法并行化。任务是 TBB 计算的逻辑单元，在应用程序中是基于 TBB 库中的 `task` 类创建的一个对象实例，而不是由操作系统管理的对象。这样应用程序中的计算调度问题就转变为对任务的调度。TBB 在语言层面提供了两种划分任务粒度的方法：指定 `grainsize`（粒度粗细）或者利用划分器，从而使 TBB 既支持用户显式定义数据的分布也支持隐式的数据分布，优化程序的局部性，提高程序性能。

TBB 任务调度器运用了 Cilk 的任务窃取调度技术来保障负载平衡。TBB 里的任务调度是“不公平的”。操作系统中线程调度典型的做法是分发时间片，这种分发是“公平的”，因为

[4] 筛选器或性能分析器

这是一个在不知道程序的高级别组织形式下最安全的策略。基于任务编程时，任务调度有应用的高层信息，所以可以为了效率而牺牲公平性。实际上，它经常延迟启动一个任务直到进程确实要用到它为止。

TBB 使用细粒度锁或无锁（lock-free）技术等方法实现并发容器。这样多个线程可以安全地并发访问共享数据，同时能得到并行加速比。但并发容器是有代价的，比常规 STL 容器开销高。为了解决并发环境中内存分配的瓶颈，TBB 中每个线程有一个内存分配器，代替 malloc/realloc/free 函数调用和 new/delete 操作。TBB 提供两种内存分配模板，scalable_allocator<T>和 cache_aligned_allocate<T>。TBB 的同步原语主要包括互斥执行和原子操作。在 TBB 中，互斥执行是由互斥体和锁实现的。互斥体是一个对象，线程可以获得互斥体上的锁，但每次只能有一个线程获得，其他线程必须等待锁被释放。TBB 中用原子操作代替部分互斥执行，atomic<T>用机器指令实现原子操作。

```
class SumFoo {
    float* my_a;
public:
    float my_sum;
    void operator() (const blocked_range<size_t>& r) {
        float *a = my_a;
        float sum = my_sum;
        size_t end = r.end();
        for (size_t i=r.begin(); i!=end; i++)
            sum += Foo(a[i]);
        my_sum = sum;
    }
    SumFoo (SumFoo& x, split) : my_a(x.my_a), my_sum(0) {}
    void join (const SumFoo& y) {my_sum += y.my_sum;}
    SumFoo (float a[]) : my_a(a), my_sum(0) {}
};

float ParallelSumFoo (const float a[], size_t n)
{
    SumFoo sf(a);
    parallel_reduce (blocked_range <size_t> (0, n), sf);
    return sf.my_sum;
}
```

图4. TBB 实现的并行求和

TBB 是按 C++的模板库形式给出所有接口函数的，容易被 C++程序员接受，丰富的并行算法和并发容器也使其具有较高的编程层次。TBB 基于任务表示，能表达任务和数据两种并行模式，适用于共享内存编程的多种应用。TBB 的动态任务调度具有好的性能潜力。TBB 在 2006 年推出，至今还在不断完善，正在走向成熟。英特尔的 ArBB[5]的运行时系统是用 TBB 写的。TBB 并且作为 Intel Parallel Studio（并行编程工具）和 Intel Threading Tools

5 Array Building Block，英特尔最新推出的并行编程技术

（线程工具）的一部分，提供一整套并行程序解决方案。

4.2 Sequoia/Sequoia++

Sequoia编程模型在2006年左右由斯坦福大学提出，其设计目标是使用户能更方便地编写存储层次敏感的并行应用程序，并使其在不同存储层次的计算平台之间具有性能的可移植性。Sequoia最初是在C语言基础上扩展而成，后来斯坦福在Sequoia基础上做了进一步语言扩展，包含了一些C++的语言特性，并改称为Sequoia++。因为两者在并行扩展部分几乎一致，所以后面统称为Sequoia[16][18]。目前，Sequoia编程模型支持的平台包括多种多核/众核平台及其集群，例如Cell、PS3、通用多核、Roadrunner、集群[19]。此外，Sequoia将来也会支持通用GPU平台[22]。

Sequoia编程模型的机器抽象是一棵存储层次树[20]。存储层次抽象树的每一层可能是托管、共享或者虚拟的存储。托管表示这一层由操作系统管理，例如硬盘；共享表示处理单元可以利用这一层进行共享通信；虚拟存储不对应于具体的物理存储体，只是将子节点统一起来。这样，Sequoia既支持垂直通信，也能支持同层节点之间的横向通信

Sequoia的核心原则就是让程序员控制程序中的局部性和通信。具体的方法就是把这两个因素都包装在一个任务中，并把源代码与机器映射相关信息相分离，让程序员实现对于局部性和通信敏感的算法。Sequoia鼓励采用分治法求解应用。Sequoia程序可以看成是一个任务的集合。每个任务是一个函数，函数只能访问其参数或者其局部数据。这保证了Sequoia不同任务的访存空间是相互隔离的，防止发生数据竞争。

```
void task matmul::inner ( in float A[M][P], in float B[p][N], Inout float C[M][N] ){
  //  Tunable parameters specify the size of subblocks of A, B, and C.
  tunable int U;  tunable int X;   tunable int V;
  //  partition matrices into sets of blocks using regular 2D chopping.
  blkset Ablks = rchop( A, U, X );
  blkset Bblks = rchop( B, X, V );
  blkset Cblks = rchop( C, U, V );
  // compute all blocks of C in parallel
  mappar( int I = 0 to M/U, int j = 0 to N/V ) {
    mapreduce ( int k = 0 to P/X ) {
      // Invoke the matmul task recursively on the subblocks of A, B and C
      matmul ( Ablks[i][k], Bblks[k][j], Cblks[i][j] ); }  }   }
```

图5. 矩阵乘的Sequoia程序示例

图5是一个Sequoia的矩阵乘示例。Tunable变量是体系结构相关的重要变量，其数值是影响应用性能的重要因素。Tunable变量从机器映射文件得到其数值，在编译时作为常数被使用。机器映射文件定义任务到各个存储层上的映射，其中包括Tunable变量、数据拷贝等存储调度上的优化操作制导（例如，是否需要拷贝数据、是否采用双缓冲等），机器映射文件为用户提供了一个针对具体硬件平台优化程序性能的手段。这样Sequoia程序在

层次存储敏感的体系结构上能追求跨平台的最优性能，并具有较好的可编程性。对于数据并行，Sequoia 提供了 `rchop`、`ichop` 与 `gather` 等数据分布手段，用 `mappar`、`mapreduce` 与 `mapseq` 等表达循环并行[20]。Sequoia 还提供两种同步[21]，分别是 `WaitTask` 与 `Barrier` 函数。前者是父任务等待子任务；后者是在隶属于同一父任务的某个子任务集合上的同步，并要求该子集内的线程编号必须是连续的。

Sequoia 最近增加了对不规则应用的任务并行的支持[23]， `call-up` 支持子任务调用在其父任务上的方法并在父任务的地址空间上完成该计算，`spawn` 支持任务派生。因此，Sequoia 除了数值计算领域，也开始支持动态非规则的应用。

Sequoia 通过将存储层次暴露给用户，能够适应多核平台上存储层次不断增加的趋势，可以很好地挖掘机器性能。在正确性方面，Sequoia 需要用户指定任务所需要的数据集并只支持垂直通信，使得子任务之间相互隔离，避免了数据竞争。在编程效率与可移植性方面，递归分治的方法对应用程序的编写有一定的局限性。Sequoia 源程序是机器无关的，程序到机器之间的映射是通过机器映射文件来决定的，并可以在该文件中给出通信优化制导、数据拷贝/分块参数来优化性能[20]，因此移植一个 Sequoia 程序到不同的平台，只需要重新编写机器映射文件，具有较好的可移植性。

综上所述，Sequoia 便于在存储层次明显的平台上编写高性能的并行程序，具有较好的可移植性，但是其应用的适用面还比较局限。目前，Sequoia 已经有一个实验性质的编译器与运行时环境。

5 异构平台的编程框架

通用 GPU 等众核处理器的发展引起用户的很大关注。各种编程接口和编程系统应运而生，比如 NVIDIA 的 CUDA（已经同时支持 C 语言和 Fortran）、英特尔推出的 ArBB[26]、专门针对流式应用的 StreamIT 以及 AMD 公司的 Brook+、PGI 提出的高层编程接口 PGI Accelerator[2]和 CAPS 公司的 HMPP[3]。其中，HMPP 和 PGI Accelerator 都是基于 C/Fortran 的语言制导的扩展，适合遗产代码到异构平台的迁移，允许用户指出哪部分计算在加速器上执行（映射），允许用户描述针对加速器的数据移动（分配、移入、移出，以及冗余移动的删除）和调度优化，并通过指定线程分组拓扑、循环属性和循环变换的细节等参数优化加速器的生成代码。在这些编程接口中，OpenCL 作为产业界的标准而受到更多的关注。

5.1 OpenCL

开放计算语言 OpenCL (Open Computing Language)是面向包括 CPU、GPU、Cell、DSP 等多核、众核处理器的异构计算平台，针对通用计算的一个开源、免费的并行编程标准。它由苹果公司首先发起，之后由开发组织 Khronos 负责协调制定 OpenCL 规范、架构等标准。目前业界主要的图形、嵌入式和桌面计算相关的硬件厂商都是 OpenCL 的成员。OpenCL 是基于 C99 的扩展，2008 年底推出语言规范 1.0 版，2011 年 11 月推出了语言规范的 1.2 版。OpenCL 刚刚推出不久，就得到了业界很多公司，如 AMD、苹果、NVIDIA、诺基亚、VIA（视频处理芯片）和 IBM（Power 芯片）等的支持。苹果的下一代操作系统 Mac OS X Snow Leopard 和重要的移动图形及视频处理器 PowerVR 已宣布要支持 OpenCL。

OpenCL 的机器抽象包含一个主机和一个以上的 OpenCL 设备。后者具有层次的组织结构；第一层是计算单元（CU 或者 compute units）；再进一步分为更多的处理单元（PE 或者 processing elements）。主设备以命令提交的形式，让设备内的 PE 执行对应的计算。CU 内

的 PE 共享指令流（锁步执行），或者每个 PE 作为一个单程序流多数据流执行单位。

OpenCL 程序包括了主机程序和 Kernel 程序，前者规定后者的设备上下文并控制后者的执行。Kernel 定义在一个 N 维索引空间（NDRange）上，该空间上的每个点的计算对应一个 kernel 实例（`work-item`），多个 `work-item` 分组构成 `work-groups`，从而可发掘层次的数据并行性（图 6 给出了向量点积的 GPU 端代码）。OpenCL 的任务并行是指每个 kernel 都不与任何其它索引空间有依赖关系，用户通过使用向量数据类型，或者将任务压入设备（device）的任务队列来实现任务并行。任务的识别和任务到设备的映射需要程序员静态完成。

OpenCL 实现的是一种放松一致性、共享存储的模型[24]。对设备的存储层次做了统一的抽象，比如 kernel 可以访问三个层次上的私有（Private）存储、局部（local）存储、常量（constant）存储、全局（global）存储。用户需要显式地声明数据的存储类型，显式地在不同存储体之间移动数据。同组的任务可以通过局部存储器和全局存储器进行栅障（barrier）同步。不同组的任务不能在同一个 kernel 内通信，只能通过全局存储器通信。

```
__kernel void DotProduct (__global float* a, __global float* b,
                                __global
                             float* c, int iNumElements){
// find position in global arrays
int iGID = get_global_id(0);
if (iGID >= iNumElements){  // bound check
    return; }
int iInOffset = iGID << 2;
c[iGID] = a[iInOffset] * b[iInOffset]
        + a[iInOffset + 1] * b[iInOffset + 1]
        + a[iInOffset + 2] * b[iInOffset + 2]
        + a[iInOffset + 3] * b[iInOffset + 3];
}
```

图6. 向量点积的 GPU 端代码

OpenCL 提供了一个运行时的支撑库[25]，程序员通过调用相关的应用程序接口（API）函数实现 CPU 与 GPU 间的同步和数据传输、GPU 显存上的存储管理、设备上下文管理、kernel 调用等。

OpenCL 标准制定的时间还比较短，标准仍在不断的完善中，比如还没有考虑 Fortran 的支持。OpenCL 作为一个标准，其未来的发展趋势取决于业内参与的厂商。NVIDIA 对 CUDA 的工具链支持已经非常成熟。由于 OpenCL 与 CUDA 的相似度非常高，两者存在竞争性。如果 OpenCL 在多种不同的异构平台上都有比较好的支持，作为商业标准的 OpenCL 就会具有很好的前景。

6 高并发的函数式语言——Erlang

Erlang 语言诞生于 1986 年，最初是由爱立信公司设计，用于电信产品的快速开发，支持大型、软实时、可移植、高并发、分布式、容错、持续运行和非停机的代码热替换应用系统的构建，适合开发运行在多核集群上的高伸缩性的系统。随着 Erlang 虚拟机性能的不断

提升，1997年Erlang已成为适合编写大规模工业产品的语言。1998年起，Open-source Erlang[27]开源组织发布开源版本，目前版本是 R14B。与此同时，Erlang 编程工具和领域函数库随之发展，为用户提供了良好的编程环境。引文[28]预言 Erlang 将成为一个非常重要的语言，也许就是下一代的 Java 语言。

Erlang 语言继承了 Prolog 语言的函数式特征以及 EriPascal 和 Ada 语言的并发、分布和异常处理机制。并行部分遵照基于消息传递实现并发驱动的 Actor 模型[29][6]。Erlang 程序以字节码在虚拟机中解释执行，具有跨平台性。Erlang 适合电信、WEB 服务、软实时数据库等应用的开发，但不适合以性能为首要需求的应用，例如图像处理和信号处理等。

Erlang 的并发执行单元为特殊的轻量进程，而不是操作系统的进程和线程。程序员通过派生轻量级进程来指定并发执行的任务。Erlang 进程调度的上下文切换的开销低，保证了高并发情况下 Erlang 应用的运行效率。

在降低多核编程复杂性的方面，作为函数式语言，Erlang 进程并发执行期间，数据访问区只包括私有栈和私有堆，这完全消除了程序员为确保正确性而对共享内存访问进行加锁/解锁的麻烦。Erlang 进程之间唯一的通信方法是消息传递，不采用共享内存的方式，只要知道一个进程的名字（pid），就可以向其发送消息。进程也可以在任何时候接收消息。这种朴素的做法让系统更加简单，也有良好的性能扩展。另外，Erlang 程序中的变量只能单次赋值，这便于调试程序错误也避免变量共享带来的锁操作。Erlang 编程中，程序员无需考虑并行编程中最容易出错的和复杂的问题，包括锁、互斥量、信号量和同步原语等，负载平衡和消息机制的实现也完全由运行时系统负责。

实时性方面，Erlang 的调度考虑了公平性,不允许某个进程长时间阻塞机器。Erlang 采用的时间受限的垃圾自动回收技术也不会长时间阻塞系统。Erlang 异常处理机制可以监视表达式的计算、进程的行为和未定义函数的调用等。 Erlang 的 Link 和 Monitor 机制将进程连接起来。某个进程出错或退出时，其它进程能感知。Erlang 虚拟机允许程序代码在运行时能够得到更改，旧的代码块被新的代码块替代。在新旧交替之际，两个版本的代码可以同时运行，这给在线软件纠错和升级提供了可能，从而保证满足控制系统持续运行的需求。

```
-module(geometry).
-export([area/1]).

area({square, Side}) ->
    Side * Side;
area({circle, Radius}) ->
    math:pi() * Radius * Radius;
area({triangle, A, B, C}) ->
    S = (A + B + C)/2,
    math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) ->
    invalid_object.
```

图7. 模式匹配代码示例

示例. 与过程式语言不同，Erlang 在编程风格上是声明式的。图 7 展示了这一编程风格。示例中，避免使用分支语句，转而采用模式匹配实现控制流的转移，运行时系统会根据形参和实参的匹配关系自动选择相应的函数执行。

Erlang 语言发展至今已是一门成熟的语言，在产业界和学术界都存在实际的应用。Open-source Erlang 开源组织负责 Erlang 系统的维护、技术支持和新版本发布，为程序员提

[6] 在计算机科学中， Actor 模型是并发计算的数学模型，是若干并发系统实现的理论基础。actor 作为基本的并发计算对象，对收到的消息作出响应，并局部决策如何对下一条收到的消息作出响应。也能创建更多的 actors，发送更多的消息

供了丰富的编程工具和函数库等。

产能方面，对于有函数式语言编程背景的程序员，在涉及分布式和容错处理的系统开发中，用 Erlang 所花的时间比用 C 少 80%，相比于命令式语言，用声明式语言编写以控制为主的电信应用更加容易，代码行数更少[30]。Open-source Erlang 为程序员提供了丰富的编程工具和函数库。Erlang 的变量单次赋值、静态/动态类型检查、异常处理机制、垃圾自动回收、进程的连接和监视都增强了应用程序的正确性保证。

借助先天的语言特性和实现机制，Erlang 在大规模分布式并发控制、错误检查和容错、代码热替换和快速开发软实时控制系统方面有明显的优势，其可移植性、使用的广泛性和开源资源方面也有不错的表现。由于 Erlang 是通过中间码解释执行的，其性能逊于以本地代码（native code）直接执行的语言，通过将 Erlang 程序中性能关键的部分以本地代码执行或改用其它语言编写能解决性能问题。Erlang 的进程由程序员显式控制，保持灵活的同时也给程序员带来了负担。

7　PGAS 语言

分割全局地址空间（PGAS）语言出现于 1998、1999 年，是一种混合的编程模型，与共享内存模型很接近。PGAS 中，共享的数据空间在各个线程之间进行逻辑的划分（通过显式的数据分布），因此从一个线程的角度，共享数据又分为本地的共享数据和远程共享数据。PGAS 抽象出计算平台上的访存不均匀性，让用户描述计算和任务的亲和关系，以提高性能的可扩展性。在可编程性方面，全局地址空间的使用简化了用户编程，远程的共享数据访问被翻译成通信，具体地调用底层的单边通信库，比如 SHMEM、GASNet、ARMCI、DCMF 或 LAPI。静态分割的全局地址空间语言具有单程序流多数据流的并行模式，包括：UPC、Co-array Fortran 和 Titanium。异步全局分割地址空间编程模型（APGAS），是在单程序流多数据流执行模型之外，增加了细粒度的任务并行的支持，包括 IBM 的 X10，克雷（Cray）的 Chapel 等语言。

下面我们分别介绍典型的静态 PGAS 语言和异步 PGAS 语言。

7.1 UPC

UPC 是一个广为受到关注和研究的 PGAS 语言。UPC 的全局地址空间、共享指针的设计思想来自于 split-c，后者是 parallel C 的变种。UPC[34]语言主要针对传统的高性能计算，支持共享存储平台或者分布式存储平台。UPC 最早的语言规范 V0.9 在 1999 年 5 月份发表，在 2000 年的 UPC Workshop 上公布了语言规范 V1.0，并在 2003 年完成语言规范 V1.1。最近的语言规范是 V1.2，发布于 2005 年。

UPC 的存储空间分为私有和共享两部分。共享空间被划分为多个分段，每个分段亲和在一个 UPC 线程上。UPC 语言支持用户定义数组的分布（方式），让用户控制程序的局部性。每个 UPC 线程除了具有一个共享空间的分段，还具有自己的私有数据空间。UPC 中对共享空间的访问均通过共享指针进行，如图 8 所示。

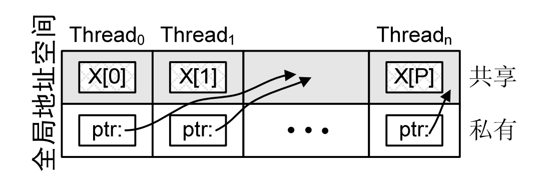


图8. UPC 共享地址空间示意图

存储一致性模型方面，UPC 提供顺序一致性和放松的一致性两种模型。前者是为方便

编程但是性能差。对共享数据，UPC 提供静态声明和动态申请两种方式。共享指针是有结构的胖指针，UPC 提供了基本的地址操作符。胖指针的开销较大，UPC 提供了强制类型转换，把数据亲和在本地的胖指针转化为普通的 C 指针。在同步支持方面，除了阻塞式的全局栅障，也提供分阶段的全局栅障以支持计算和通信的重叠，但是没有点到点的同步。在计算划分方面，UPC 提供了 `forall` 结构，通过引入亲和表达式，指导在一个全局视角的迭代空间上进行计算划分。UPC 提供多种集合操作，并且有了一个并行 I/O 的扩展建议。

但是，UPC 中扁平的并行性与多核/众核的存储层次增加的趋势不协调。中国科学院计算技术研究所的先进编译技术团队提出了基于 UPC 的层次并行扩展 HUPC（Hierarchical UPC），并分别针对 GPU 集群和通用多核集群予以实现。HUPC 引入的新特性包括多维多层的数据分布定义、单程序流多数据流与 `fork-join` 混合的执行模型、线程分组和线程组同步。通过这些扩展，HUPC 能够适用更多应用，更好适应并行计算平台上的层次存储结构。

UPC 作为较早成熟的一个 PGAS 编程语言，得到众多研究结构与公司的支持，例如加州大学伯克利分校、乔治.华盛顿大学、惠普、克雷、IBM、SGI、劳伦斯・利弗莫尔国家实验室、劳伦斯伯克利国家实验室和韩国的庆上大学（GNU）等长期参与 UPC 编译和运行时系统的研制。伯克利的 UPC 小组的编译器更新速度很快，几乎每年都有两次新版本发布。UPC 目前已经有一个成熟的使用环境，既有开源的编译环境也有商业产品，还有开发工具支持，例如，GASP 与 PPW。其中，GASP 是工具开发接口，为性能工具的开发者提供了与具体的 GAS 编译器和运行时无关的开发接口[31]，PPW 用于对 UPC 程序进行性能分析。

综上所述，对于在多核/众核平台上编程，与 OpenMP 相比，UPC 能够更好地控制数据局部性；与 MPI、Sequoia 等数据空间隔离的编程模型相比，UPC 能够增加数据共享、减少对存储空间的需求；在可编程性方面，UPC 提供的全局地址空间简化了程序员的编程，而且 UPC 是基于标准 C 语言的扩展，易于学习，因此 UPC 具有较好的编程效率。但是在通信与同步方面，UPC 目前还不支持线程分组，因此不支持分组同步或者集合通信，所以在某些应用中可能会带来额外的同步或者通信代价；在并行性表达方面，UPC 目前支持静态任务并行，缺乏负载平衡的支持；另外，受具体实现的限制，为获得较好性能，需要用户手工参与较多优化，一定程度上影响了 UPC 的编程效率。

7.2 X10

X10 是美国国防部高级研究计划局（DARPA）在高性能计算领域的 HPCS[7]计划的产物，目标是解决千万亿次（Peta）规模计算系统上应用开发中的 4 个问题[37]：(1). 安全性、(2). 可分析、(3). 性能扩展性、(4). 灵活性，但后来并不局限于科学计算领域。目前 X10 的开源编译环境版本号是 V2.1.2，支持 Eclipse 的可视化开发界面。该编译环境可以运行在主流的操作系统平台上，例如 windows、AIX、Linux 与 MacOS 等操作系统，支持 Power、X86、X86_64 与通用 GPU 等硬件平台。X10 最初主要由 IBM 研究人员开发，后来，有很多高校及研究结构人员参与相关工作[35]。目前，X10 的语言规范在每年都会有一个较大版本更新，例如，在 2009 年 11 月份推出 2.0 版本，2010 年 10 月份推出 2.1 版本。

X10 是在类 Java 语言上的并行扩展，属于异步全局分割地址空间编程模型（APGAS）。在 X10 语言中，库所(place)是个核心概念，它用于表达系统中的访存不均匀性，是若干驻留的轻量线程（也称为活动，activity）和数据的整体。库所必须被映射到缓存一致的计算单元上（比如 SMP 节点），其上有很多活动（activity）。X10 同时支持单程序流多数据流和并行语句块的执行模型，并且支持动态任务派生。这些特征使得 X10 可以从传统的数值计算拓

[7] High Productivity Computing Systems，高生产率计算机系统

展到商业服务器应用。在 X10 中，用户可以通过 ateach 或者 async 关键字动态派生新的活动。例如， ateach 语句能够将任务按照指定的数组元素的亲和性（或者一个索引集映射）分配给对应的库所；async 则支持细粒度动态任务，以及计算与通信的重叠。

```
def run()
{
  finish async{
    val c = clock.make();
    val D_Base = Dist.makeUnique(D.places());
    val diff = Array_make[Real] (D_Base),
       scratch = Array.make[Real] ( D_Base );
    ateach ( z in D_Base ) clocked(c)
      do {
        diff( z ) = 0.0;
          for( p in D | here ){
            Temp(p) = A(stencil_1(p)).reduce(Double.+, 0.0) / 4;
            diff(z) = Math.max( diff(z), Math.abs(A(p) – Temp(p)));
          }
          next;
          A(D | here) = Temp(D | here);
          reduceMax(z, diff, scratch);
        } while ( diff(z) > epsilon );
    }
}
```

图9. 热传递应用程序示例

图 9 的 X10 程序中，根活动创建了一个同步钟 c，数组 diff 与 scratch 被分布到既定的 place 空间上。然后以单程序流多数据流的风格对分布在本地的数据进行 stencil 计算和本地归约，用同步钟栅障同步后进行全局归约。这个过程一直迭代下去直至精度得到满足

目前，X10 并不支持任务的动态负载平衡。论文[39]在 X10 上扩展了适应性的任务调度策略，在 work-first 与 help-first 两种任务窃取策略中选择。文章[38]借鉴了 Sequoia 的部分概念，提出了层次化的 place 的概念(HPT, Hierarchical Place Trees)，除了数据分布隐含的垂直通信，HPT 还支持显式的同步和异步数据传输。这样，HPT 程序能适应通用 GPU 加速器，也能更好地映射到存储层次增加的普通多核平台。

在正确性方面,根据引文[37]，用 X10 编写的程序能够保证静态类型安全、存储安全与指针安全。同时，X10 提供了对异常处理的支持。在死锁与数据竞争方面，X10 设计了同步钟 clock，用户只要遵循简单的语法规则就可以避免死锁与数据竞争，即只使用 async, finish, at，atomic，clock 的 X10 程序不会导致死锁。

综上所述，X10 在并行编程模型方面有不少技术亮点引起学术界的关注。但是由于语言庞大，编译系统的开发耗时，影响了其推广。其次，X10 语言是基于类 Java、C++的扩展，虽然其串行部分类似于 C，对于熟悉 C 与面向对象的用户来说便于学习，但是毕竟它属于一个新语言，这对于其推广也是不利的。如何能适应更多的应用领域，并实现与现有语言的良好的互操作，是影响 X10 发展的重要因素。

8 结束语

本文介绍了多种流行的并行编程模型，我们可以看到这样的趋势：(1). 应用驱动变革。片上并行系统的出现，对桌面和商业应用领域影响最深远。应用行为特征的变化，使得编程语言和编译技术必须要处理更不规则的并行性。高吞吐的应用需求也催生了像 Erlang 这样的编程语言。(2). 可编程性是业界的追求。在芯片的多核时代，程序员仍然希望能够享受"免费的性能午餐"，因此最受关注的就是自动并行化技术。但是我们看到基于制导的 OpenMP 语言在对大型实际应用的并行化上仍存在较大问题，DSWP 模型近年来虽颇为活跃，但是由于还无法通过下载得到该模型可供使用的编译器，而且相关文章的很多实验均为手工完成，再考虑到实际的程序分析技术的能力，该模型的实际效果目前还有待证实。在并行语言的设计中，引入面向对象、泛型等现代语言特征，引入函数式语言特征，把存储和计算的逻辑单位和物理单位进行分离，提供高层的并行结构和语言机制，提供共享内存的存储抽象都是提高可编程性的重要手段。(3). 在多核的复杂存储和互连结构上，数据移动是程序优化的重点。MPI、PGAS 和 OpenCL 把数据移动的优化交给用户。OpenCL 可以统一异构平台的节点内编程，只是编程层次较低。PGAS 语言能更好地控制数据和计算的亲和性，适合编写高性能的支撑软件，具有统一集群编程的潜力。而像 TBB 这类面向大众的编程语言也通过任务调度和隐式数据分布的结合来支持数据重用。(4). 复杂的多核、众核结构上的跨平台的性能调优受到关注。在这里，机器抽象具有重要的意义，编程模型也需要支持自动或手工的性能调优。Sequoia 语言基于存储层次树的机器抽象，把源程序与机器映射文件分离，提供了跨平台的性能可移植性。类似的思想也被用到 Habanero(X10)的 HPT 扩展上。

可以设想，未来的并行程序设计一定是多种模型和语言共存的状态，每一种模型和语言有自己适用的应用类型和体系结构。在这场变革中，对于每一个参与者都是机会与挑战并存。除了编程模型本身的技术优势和应用本身的驱动以外，遗产代码和商业推动也是不容忽视的因素。

参考文献：

[1] Eduard Ayguadé, Rosa M. Badia, etc. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *Int J Parallel Prog* (2010) 38:440–459

[2] The Portland Group. PGI Fortran & C Accelerator Programming Model[OL]. http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.2.pdf

[3] HMPP Workbench Build manycore applications. http://www.cse.scitech.ac.uk/disco/workshops /200907/HMPPWorkbench-Tutorial-2009.pdf

[4] Eduard Ayguade, James Beyer, etc. An Extension to Improve OpenMP Tasking Control. IWOMP2010

[5] Eduard Ayguade, Nawal Copty, etc. The Design of OpenMP Tasks. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, VOL. 20, NO. 3, MARCH 2009

[6] Matthew Bridges, Neil Vachharajani, Yun Zhang, etc., Revisiting the Sequential Programming Model for Multi-Core, presented at the Proceedings of *the 40th Annual IEEE/ACM International Symposium on Microarchitecture* [C], 2007.

[7] Neil Vachharajani, Ram Rangan, Easwaran Raman, etc., Speculative Decoupled Software Pipelining, presented at the Proceedings of *the 16th International Conference on Parallel Architecture and Compilation Techniques [C]*, 2007.

[8] http://meetings.mpi-forum.org/MPI_3.0_main_page.php

[9] H. Tang, K. Shen, and T. Yang. Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines. ACM Transactions on Programming Languages and Systems, 2000. An earlier version appeared in Proc. of *ACM Symposium on Principles & Practice of Parallel Programming (PPoPP)*, 1999. Pages 107-118. 1999.

[10] Hong Tang and Tao Yang. Optimizing threaded MPI execution on SMP clusters. In Proceedings of the 15th international conference on Supercomputing (ICS '01). *ACM, New York,* NY, USA, 381-392. 2001.

[11] Ron Brightwell and Kevin Pedretti, "Optimizing Multi-core MPI Collectives with SMARTMAP"，Parallel Processing Workshops, 2009. ICPPW '09

[12] A.R.,Kumar, R.,De, D.,Panda, D.K, "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics", Mamidala,.,Dept. of Comput. Sci. & Eng., Ohio State Univ., Columbus, OH,Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on

[13] http://www.multicore-association.org；

[14] Damian A. Mallon, "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures"，*EuroPVM/MPI 2009*, LNCS 5759, pp. 174–184, 2009.

[15] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. Toward Efficient Support for Multithreaded MPI Communication

[16] James Reinders. *Intel Threading Buling Blocks*. O'REILLY Media. 2007.12.

[17] http://www.stanford.edu/group/sequoia/cgi-bin/node/32

[18] http://www.stanford.edu/group/sequoia/

[19] http://ppl.stanford.edu/wiki/index.php/Sequoia

[20] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, H. Park, M. Ren, A. Aiken, W. Dally and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In Proceedings of Supercomputing 2006, November 2006.

[21] M. Houston, J.-Y. Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan. A Portable Runtime Interface for Multi-Level Memory Hierarchies. Proceedings of the Symposium on Principles and Practice of Parallel Programming, pages 143-152, February 2008.

[22] Michael Bauer, John Clark, Eric Schkufza, Alex Aiken. A CUDA Runtime Target for the Sequoia Compiler. *NVIDIA Research Summit 2010*.

[23] Michael Bauer, John Clark, Eric Schkufza, Alex Aiken. Programming the memory hierarchy revisited: supporting irregular parallel in sequoia. Proceedings of the *Symposium on Principles and Practice of Parallel Programming*, pages 13-23, February 2011.

[24] OpenCL Specification[OL]. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[25] Khronos OpenCL API Registry[OL]. http://www.khronos.org/registry/cl/

[26] http://software.intel.com/sites/products/documentation/arbb/arbb_userguide_linux.pdf

[27] Open Source Erlang[OL], 2009-10-13, http://www.erlang.org.

[28] 开源 Erlang 真的能成为下一代 Java 语言吗，赛迪网，2009-07-22，http://tech.ccidnet.com。

[29] Carl Hewitt; Peter Bishop and Richard Steiger (1973). A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI.

[30] Joe Armstrong, The development of Erlang, Proceedings of the second ACM SIGPLAN international conference on Functional programming, p.196-203, June 09-11, 1997, pages 196 – 203, http://www.erlang.org/doc.html.

[31] Hung-Hsun Su, Dan Bonachea, Adam Leko, Hans Sherburne, Max Billingsley III, Alan D. George. GASP! A Standardized Performance Analysis Tool Interface for Global Address Space Programming Models. Lawrence Berkeley National Lab Tech Report LBNL-61659, 2006.

[32] Francois Cantonnet,Tarek A. El-Ghazawi, Pascal Lorenz, Jaafer Gaber. Fast Address Translation Techniques for Distributed Shared Memory Compilers. Proceedings of *the 19th IEEE International Parallel and Distributed Processing Symposium* (IPDPS'05). Pages: 52.2

[33] C. Bell, D. Bonachea, R. Nishtala, K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. *20th International Parallel & Distributed Processing Symposium (IPDPS)*, 2006.

[34] UPC Consortium. UPC Language Specifications, v1.2. *Lawrence Berkeley National Lab Tech Report* LBNL-59208, 2005.

[35] http://x10plus.cloudaccess.net/x10-community/universities-using-x10.html

[36] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, David Grove.X10 Language Specification Version2.1.2011.

[37] Philip Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, Vivek Sarkar. X10: An Object-oriented approach to non-uniform Clustered Computing. OOPSLA 2005.

[38] Yonghong Yan, Jisheng Zhao, Yi Guo, Vivek Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In Proceedings of LCPC'2009. pp.172~187.

[39] Yi Guo, Rajkishore Barik, Raghavan Raman, Vivek Sarkar. Work-First and Help-First Scheduling Policies for Terminally Strict Parallel Programs. *23rd IEEE International Parallel and Distributed Processing Symposium* (IPDPS), May 2009.

作者简介：

陈　莉： 中国科学院计算技术研究所系统结构重点实验室，副研究员，lchen@ict.ac.cn

霍　玮： 中国科学院计算技术研究所系统结构重点实验室，助理研究员

卢兴敬： 中国科学院计算技术研究所系统结构重点实验室，工程师

唐生林： 中国科学院计算技术研究所系统结构重点实验室，工程师